25장 클래스

25.1 클래스는 프로토타입의 문법적 설탕인가?

- 자바스크립트는 프로토타입 기반의 객체지향 언어
- 본래는 클래스가 필요 없는 (Class free) 객체지향 프로그래밍 언어

```
// ES5 생성자 함수
var Person = (function () {
  // 생성자 함수
  function Person(name) {
    this.name = name;
  }

  // 프로토타입 메서드
  Person.prototype.sayHi = function () {
    console.log('Hi! My name is ' + this.name);
  };

  // 생성자 함수 반환
  return Person;
}());

// 인스턴스 생성
var me = new Person('Lee');
me.sayHi(); // Hi! My name is Lee
```

- 하지만 클래스 기반 언어에 익숙한 프로그래머들에게 높은 진입 장벽이었고,

  때문에 클래스 기반 객체지향 프로그래밍 언어와 매우 흡사한 매커니즘이 ES6부터 제시되었다.

- 사실 클래스는 함수이며, 기존 프로토타입 기반 패턴을 클래스 기반 패턴처럼 사용할 수 있는

  문법적 설탕 (Syntactic sugar)

클래스와 생성자 함수의 차이점

1. 클래스는 new 연산자 없이 호출하면 에러 발생.

   생성자 함수는 new 연산자 없이 호출하면 일반 함수로서 호출

2. 클래스는 extends와 super 키워드로 상속 가능. 생성자 함수는 불가능.
3. 클래스는 호이스팅이 발생하지 않는 것처럼 동작. 생성자 함수는 호이스팅이 발생.
4. 클래스 내부에는 자동으로 strict mode가 지정되며 해제할 수 없다. 생성자 함수는 그렇지 않다.
5. 클래스의 constructor, 프로토타입 메서드, 정적 메서드는 모두 열거 불가능.
   (`[[Enumerable]]` ⇒ `false`)

25.2 클래스 정의

- **class** 키워드를 사용하여 정의 가능.
- 클래스 이름은 파스칼 케이스를 사용하는 것이 일반적

  (사용하지 않는다고 에러가 발생하진 않는다)

```
// 클래스 선언문
class Person {}
```

```
// 익명 클래스 표현식
const Person = class {};

// 기명 클래스 표현식
const Person = class MyClass {};
```

- 클래스는 표현식으로 정의 가능하며 값으로 사용할 수 있는 일급 객체.
- 특징
  - 무명 리터럴로 생성 가능. 즉, 런타임에 생성이 가능.
  - 변수나 자료 구조에 저장 가능.
  - 함수의 매개변수에게 전달 가능.
  - 함수의 반환값으로 사용 가능.

```
// 클래스 선언문
class Person {
  // 생성자
  constructor(name) {
    // 인스턴스 생성 및 초기화
    this.name = name; // name 프로퍼티는 public하다.
  }

  // 프로토타입 메서드
  sayHi() {
    console.log(`Hi! My name is ${this.name}`);
  }

  // 정적 메서드
  static sayHello() {
    console.log('Hello!');
  }
}

// 인스턴스 생성
const me = new Person('Lee');

// 인스턴스의 프로퍼티 참조
console.log(me.name); // Lee
// 프로토타입 메서드 호출
me.sayHi(); // Hi! My name is Lee
// 정적 메서드 호출
Person.sayHello(); // Hello!
```

클래스의 정의 방식 vs 생성자 함수의 정의 방식

```
// 생성자 함수
var person = (function() {
    function Person(name) {
        this.name = name;
    }
...
```

```
// 클래스의 생성자
class Person {
    constructor(name) {
        this.name = name;
    }
...
```

```
// 생성자 함수 프로토타입 메서드
...
    Person.prototype.sayHi = function() {
        console.log('Hi! My name is ' + this.name);
    };
...

// 클래스의 프로토타입 메서드
...
sayHi() {
    console.log('Hi! My name is ' + this.name);
};
...
```

```
// 생성자 함수의 정적 메서드 및 함수 반환
...
    Person.sayHello = function() {
        console.log('Hello!');
    };

    return Person;
}());

// 클래스의 정적 메서드
...
static sayHello() {
    console.log('Hello!');
```

```
}
...
```

25.3 클래스 호이스팅

```
// 클래스 선언문
class Person {}

console.log(typeof Person); // function
```

- 클래스 선언문으로 정의된 클래스는 런타임 이전에 먼저 평가되어 함수 객체를 생성
  - 이때 생성되는 함수 객체는 contructor 생성자 함수
- 생성자 함수로서 호출할 수 있는 함수는 함수 객체가 생성되는 시점에
  프로토타입도 더불어 생성

```
console.log(Person);
// ReferenceError: Cannot access 'Person' before initializa
tion

// 클래스 선언문
class Person {}
```

```
const Person = '';

{
  // 호이스팅이 발생하지 않는다면 ''이 출력되어야 한다.
  console.log(Person);
  // ReferenceError: Cannot access 'Person' before initiali
zation

  // 클래스 선언문
  class Person {}
}
```

- 클래스는 let, const 변수처럼 호이스팅이 발생
- 따라서 선언문 이전에 TDZ에 들어가서 접근이 불가능

25.4 인스턴스 생성

- 클래스는 생성자 함수이며 new 연산자와 함께 호출되어 인스턴스를 생성

```
class Person {}

// 인스턴스 생성
const me = new Person();
console.log(me); // Person {}
```

- 클래스의 유일한 존재 이유는 인스턴스를 생성하는 것
- 따라서 new 연산자와 함께 호출해야 한다.

```
class Person {}

// 클래스를 new 연산자 없이 호출하면 타입 에러가 발생한다.
const me = Person();
// TypeError: Class constructor Person cannot be invoked wi
thout 'new'
```

```
const Person = class MyClass {};

// 함수 표현식과 마찬가지로 클래스를 가리키는 식별자로 인스턴스를 생성해
야 한다.
const me = new Person();

// 클래스 이름 MyClass는 함수와 동일하게 클래스 몸체 내부에서만 유효한
식별자다.
console.log(MyClass); // ReferenceError: MyClass is not def
ined
```

```
const you = new MyClass(); // ReferenceError: MyClass is no
t defined
```

25.5 메서드

- 클래스의 몸체에는 0개 이상의 메서드만 선언 가능
- 클래스 몸체에서 정의할 수 있는 메서드는 constructor, 프로토타입 메서드, 정적 메서드

25.5.1 constructor

인스턴스를 생성하고 초기화하기 위한 특수한 메서드

```
class Person {
  // 생성자
  constructor(name) {
    // 인스턴스 생성 및 초기화
    this.name = name;
  }
}
```

```
// 클래스는 함수다.
console.log(typeof Person); // function
console.dir(Person);
```

- 클래스 역시 함수와 동일하게 프로토타입과 연결
- 자신의 스코프 체인을 구성
- 모든 함수 객체가 갖고 있는 prototype 프로퍼티가 가리키는

  프로토타입 객체의 constructor 프로퍼티는 클래스 자신

```
// 인스턴스 생성
const me = new Person('Lee');
console.log(me);
```

- constructor 내부에서 this에 추가한 프로퍼티는 인스턴스 프로퍼티가 된다.

```
// 클래스
class Person {
  // 생성자
  constructor(name) {
    // 인스턴스 생성 및 초기화
    this.name = name;
  }
}

// 생성자 함수
function Person(name) {
  // 인스턴스 생성 및 초기화
  this.name = name;
}
```

- constructor는 하나의 메서드로 해석되는 것이 아니라,

  클래스가 평가되어 생성한 함수 객체 코드의 일부가 된다.
- 클래스 정의가 평가되면 constructor의 기술된 동작을 하는 함수 객체가 생성
- constructor는 클래스 내에 최대 한 개만 존재

```
class Person {
  constructor() {}
  constructor() {}
}
// SyntaxError: A class may only have one constructor
```

constructor를 작성하지 않으면 암묵적으로 빈 constructor 객체가 생성.

```
class Person {}
```

```
class Person {
  // constructor를 생략하면 다음과 같이 빈 constructor가 암묵적으
로 정의된다.
  constructor() {}
```

```
}

// 빈 객체가 생성된다.
const me = new Person();
console.log(me); // Person {}
```

- constructor 내부의 this에 인스턴스 프로퍼티를 추가하여 인스턴스를 초기화 가능.

```
class Person {
  constructor() {
    // 고정값으로 인스턴스 초기화
    this.name = 'Lee';
    this.address = 'Seoul';
  }
}

// 인스턴스 프로퍼티가 추가된다.
const me = new Person();
console.log(me); // Person {name: "Lee", address: "Seoul"}
```

- 클래스 외부에서 인스턴스 프로퍼티 초기값을 전달하려면 매개 변수로 받아올 수 있다.

```
class Person {
  constructor(name, address) {
    // 인수로 인스턴스 초기화
    this.name = name;
    this.address = address;
  }
}

// 인수로 초기값을 전달한다. 초기값은 constructor에 전달된다.
const me = new Person('Lee', 'Seoul');
console.log(me); // Person {name: "Lee", address: "Seoul"}
```

- new 연산자와 함께 클래스가 호출되면, 생성자 함수와 동일하게 암묵적으로 this 인스턴스가 반환

```
class Person {
  constructor(name) {
    this.name = name;

    // 명시적으로 객체를 반환하면 암묵적인 this 반환이 무시된다.
    return {};
  }
}

// constructor에서 명시적으로 반환한 빈 객체가 반환된다.
const me = new Person('Lee');
console.log(me); // {}
```

```
class Person {
  constructor(name) {
    this.name = name;

    // 명시적으로 원시값을 반환하면 원시값 반환은 무시되고 암묵적으로 t
his가 반환된다.
    return 100;
  }
}

const me = new Person('Lee');
console.log(me); // Person { name: "Lee" }
```

- 클래스에서 this가 아닌 다른 값을 반환하는 것은 클래스의 기본 동작을 해치는 행위.

25.5.2 프로토타입 메서드

- 클래스의 prototype 프로퍼티에 메서드를 추가하지 않아도 기본적으로 프로토타입 메서드가 됨

```
class Person {
  // 생성자
  constructor(name) {
    // 인스턴스 생성 및 초기화
```

```
    this.name = name;
  }

  // 프로토타입 메서드
  sayHi() {
    console.log(`Hi! My name is ${this.name}`);
  }
}

const me = new Person('Lee');
me.sayHi(); // Hi! My name is Lee
```

- 클래스가 생성한 인스턴스는 프로토타입 체인의 일원

```
// me 객체의 프로토타입은 Person.prototype이다.
Object.getPrototypeOf(me) === Person.prototype; // -> true
me instanceof Person; // -> true

// Person.prototype의 프로토타입은 Object.prototype이다.
Object.getPrototypeOf(Person.prototype) === Object.prototyp
e; // -> true
me instanceof Object; // -> true

// me 객체의 constructor는 Person 클래스다.
me.constructor === Person; // -> true
```

- 프로토타입 체인은 클래스에 의해 생성된 인스턴스에도 동일하게 적용

25.5.3 정적 메서드

- 정적 (Static) 메서드

  인스턴스를 생성하지 않아도 호출할 수 있는 메서드

```
// 생성자 함수
function Person(name) {
  this.name = name;
}
```

```
// 정적 메서드
Person.sayHi = function () {
  console.log('Hi!');
};

// 정적 메서드 호출
Person.sayHi(); // Hi!
```

- static 키워드를 붙이면 정적 메서드를 선언 가능

```
class Person {
  // 생성자
  constructor(name) {
    // 인스턴스 생성 및 초기화
    this.name = name;
  }

  // 정적 메서드
  static sayHi() {
    console.log('Hi!');
  }
}
```

- 정적 메서드는 클래스에 바인딩된 메서드가 된다.
- 정적 메서드는 클래스 정의 이후 인스턴스를 생성하지 않아도 호출할 수 있다.
- 프로토타입 메서드처럼 인스턴스로 호출하지 않고 클래스로 호출 가능

```
// 정적 메서드는 클래스로 호출한다.
// 정적 메서드는 인스턴스 없이도 호출할 수 있다.
Person.sayHi(); // Hi!
```

```
// 인스턴스 생성
const me = new Person('Lee');
me.sayHi(); // TypeError: me.sayHi is not a function
```

25.5.4 정적 메서드와 프로토타입 메서드의 차이

1. 정적 메서드와 프로토타입 메서드는 프로토타입 체인이 다르다.
2. 정적 메서드는 클래스로 호출하고 프로토타입 메서드는 인스턴스로 호출
3. 정적 메서드는 인스턴스 프로퍼티 참조 불가능. 프로토타입 메서든느 가능

```
class Square {
  // 정적 메서드
  static area(width, height) {
    return width * height;
  }
}

console.log(Square.area(10, 10)); // 100
```

```
class Square {
  constructor(width, height) {
    this.width = width;
    this.height = height;
  }

  // 프로토타입 메서드
  area() {
    return this.width * this.height;
  }
}

const square = new Square(10, 10);
console.log(square.area()); // 100
```

- 메서드 내부의 this는 메서드를 호출한 객체에 바인딩
- 따라서 프로토타입 메서드 내부의 this는 프로토타입 메서드를 호출한 인스턴스에 바인딩
- 정적 메서드 내부의 this는 클래스에 바인딩

> this를 사용하지 않는 메서드는 정적 메서드로 정의하는 것이 좋다.

```
// 표준 빌트인 객체의 정적 메서드
Math.max(1, 2, 3);           // -> 3
Number.isNaN(NaN);           // -> true
JSON.stringify({ a: 1 });    // -> "{"a":1}"
Object.is({}, {});           // -> false
Reflect.has({ a: 1 }, 'a'); // -> true
```

25.5.5 클래스에서 정의한 메서드의 특징

1. function 키워드를 생략한 메서드 축약 표현 사용
2. 클래스에서 메서드를 정의할 때는 콤마가 불필요
3. 암묵적으로 strict mode로 실행
4. `for … in` 또는 `Object.keys` 로 열거 불가능
5. 내부 메서드 `[[Contructor]]`를 갖지 않는 non-constructor. new 연산자와 함께 호출 불가능

25.6 클래스의 인스턴스 생성 과정

1. 인스턴스 생성과 this 바인딩

new 연산자와 함께 호출하면 먼저 암묵적으로 빈 객체가 인스턴스로 생성된다.

이 인스턴스의 프로토타입은 클래스의 prototype 프로퍼티가 가리키는 객체로 설정된다.

즉, 인스턴스는 this에 바인딩

2. 인스턴스 초기화

constructor 내부 코드가 실행되어 this에 바인딩되어 있는 인스턴스를 초기화

this에 바인딩되어 있는 인스턴스에 프로퍼티 추가

constructor에 인수로 전달된 초기값으로 인스턴스 프로퍼티를 초기화

3. 인스턴스 반환

완성된 인스턴스가 바인딩된 this가 암묵적으로 반환

```
class Person {
  // 생성자
```

```
  constructor(name) {
    // 1. 암묵적으로 인스턴스가 생성되고 this에 바인딩된다.
    console.log(this); // Person {}
    console.log(Object.getPrototypeOf(this) === Person.prot
otype); // true

    // 2. this에 바인딩되어 있는 인스턴스를 초기화한다.
    this.name = name;

    // 3. 완성된 인스턴스가 바인딩된 this가 암묵적으로 반환된다.
  }
}
```

25.7 프로퍼티

25.7.1 인스턴스 프로퍼티

- 인스턴스 프로퍼티는 constructor 내부에서 정의

```
class Person {
  constructor(name) {
    // 인스턴스 프로퍼티
    this.name = name;
  }
}

const me = new Person('Lee');
console.log(me); // Person {name: "Lee"}
```

- constructor 코드가 실행되기 전에는 암묵적으로 생성된 빈 객체가 인스턴스로써 존재
- constructor 코드가 실행되어 인스턴스에 프로퍼티를 추가하면서 초기화

```
class Person {
  constructor(name) {
    // 인스턴스 프로퍼티
```

```
    this.name = name; // name 프로퍼티는 public하다.
  }
}

const me = new Person('Lee');

// name은 public하다.
console.log(me.name); // Lee
```

25.7.2 접근자 프로퍼티

- 접근자 프로퍼티는 자체적으로는 값을 갖지 않는다
- 다른 데이터 프로퍼티의 값을 읽거나 저장할 때 사용하는 접근자 함수로 구성된 프로퍼티

```
const person = {
  // 데이터 프로퍼티
  firstName: 'Ungmo',
  lastName: 'Lee',

  // fullName은 접근자 함수로 구성된 접근자 프로퍼티다.
  // getter 함수
  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  },
  // setter 함수
  set fullName(name) {
    // 배열 디스트럭처링 할당: "36.1. 배열 디스트럭처링 할당" 참고
    [this.firstName, this.lastName] = name.split(' ');
  }
};

// 데이터 프로퍼티를 통한 프로퍼티 값의 참조.
console.log(`${person.firstName} ${person.lastName}`); // U
ngmo Lee

// 접근자 프로퍼티를 통한 프로퍼티 값의 저장
// 접근자 프로퍼티 fullName에 값을 저장하면 setter 함수가 호출된다.
```

```
person.fullName = 'Heegun Lee';
console.log(person); // {firstName: "Heegun", lastName: "Le
e"}

// 접근자 프로퍼티를 통한 프로퍼티 값의 참조
// 접근자 프로퍼티 fullName에 접근하면 getter 함수가 호출된다.
console.log(person.fullName); // Heegun Lee

// fullName은 접근자 프로퍼티다.
// 접근자 프로퍼티는 get, set, enumerable, configurable 프로퍼티
어트리뷰트를 갖는다.
console.log(Object.getOwnPropertyDescriptor(person, 'fullNa
me'));
// {get: ƒ, set: ƒ, enumerable: true, configurable: true}
```

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  // fullName은 접근자 함수로 구성된 접근자 프로퍼티다.
  // getter 함수
  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  }

  // setter 함수
  set fullName(name) {
    [this.firstName, this.lastName] = name.split(' ');
  }
}

const me = new Person('Ungmo', 'Lee');

// 데이터 프로퍼티를 통한 프로퍼티 값의 참조.
console.log(`${me.firstName} ${me.lastName}`); // Ungmo Lee
```

```
// 접근자 프로퍼티를 통한 프로퍼티 값의 저장
// 접근자 프로퍼티 fullName에 값을 저장하면 setter 함수가 호출된다.
me.fullName = 'Heegun Lee';
console.log(me); // {firstName: "Heegun", lastName: "Lee"}

// 접근자 프로퍼티를 통한 프로퍼티 값의 참조
// 접근자 프로퍼티 fullName에 접근하면 getter 함수가 호출된다.
console.log(me.fullName); // Heegun Lee

// fullName은 접근자 프로퍼티다.
// 접근자 프로퍼티는 get, set, enumerable, configurable 프로퍼티
어트리뷰트를 갖는다.
console.log(Object.getOwnPropertyDescriptor(Person.prototyp
e, 'fullName'));
// {get: ƒ, set: ƒ, enumerable: false, configurable: true}
```

- getter : 인스턴스 프로퍼티에 **접근**할 때마다

  프로퍼티 값을 조작하거나 별도의 행위가 필요할 때 사용
- setter : 인스턴스 프로퍼티에 값을 **할당**할 때마다

  프로퍼티 값을 조작하거나 별도의 행위가 필요할 때 사용

25.7.3 클래스 필드 정의 제안

- 클래스 필드 (class field)

  클래스 기반 객체지향 언어에서 클래스가 생성할 인스턴스의 프로퍼티
- 클래스 몸체에서 클래스 필드를 정의하려면 this에 바인딩하면 안된다.
- this는 constructor와 메서드 내에서만 유효

```
class Person {
  // this에 클래스 필드를 바인딩해서는 안된다.
  this.name = ''; // SyntaxError: Unexpected token '.'
}
```

- this를 반드시 사용해야한다.

```
class Person {
  // 클래스 필드
  name = 'Lee';

  constructor() {
    console.log(name); // ReferenceError: name is not defin
ed
  }
}

new Person();
```

- 초기값을 할당하지 않으면 undefined.

```
class Person {
  // 클래스 필드를 초기화하지 않으면 undefined를 갖는다.
  name;
}

const me = new Person();
console.log(me); // Person {name: undefined}
```

- constructor 내부에서만 초기화해야 한다.

```
class Person {
  name;

  constructor(name) {
    // 클래스 필드 초기화.
    this.name = name;
  }
}

const me = new Person('Lee');
console.log(me); // Person {name: "Lee"}
```

- 클래스 필드에 함수를 할당하면 인스턴스의 메서드가 된다.

- 모든 클래스 필드는 인스턴스 프로퍼티이기 때문
- 따라서 클래스 필드에 함수 할당은 비권장사항이다.

```
<!DOCTYPE html>
<html><body><button class="btn">0</button><script>
    class App {
      constructor() {
        this.$button = document.querySelector('.btn');
        this.count = 0;

        // increase 메서드를 이벤트 핸들러로 등록
        // 이벤트 핸들러 increase 내부의 this는 DOM 요소(this.$b
utton)를 가리킨다.
        // 하지만 increase는 화살표 함수로 정의되어 있으므로
        // increase 내부의 this는 인스턴스를 가리킨다.
        this.$button.onclick = this.increase;

        // 만약 increase가 화살표 함수가 아니라면 bind 메서드를 사
용해야 한다.
        // $button.onclick = this.increase.bind(this);
      }

      // 인스턴스 메서드
      // 화살표 함수 내부의 this는 언제나 상위 컨텍스트의 this를 가리
킨다.
      increase = () => this.$button.textContent = ++this.co
unt;
    }
    new App();
  </script></body></html>
```

25.7.4 private 필드 정의 제안

- 인스턴스 프로퍼티는 클래스 외부에서 접근이 가능한 public

```
class Person {
  constructor(name) {
    this.name = name; // 인스턴스 프로퍼티는 기본적으로 public하
```

```
다.
  }
}

// 인스턴스 생성
const me = new Person('Lee');
console.log(me.name); // Lee
```

- #을 사용하여 private을 정의할 수 있는 표준 사양이 TC39 프로세스의 stage 3에서 제안되었다.

```
class Person {
  // private 필드 정의
  #name = '';

  constructor(name) {
    // private 필드 참조
    this.#name = name;
  }
}

const me = new Person('Lee');

// private 필드 #name은 클래스 외부에서 참조할 수 없다.
console.log(me.#name);
// SyntaxError: Private field '#name' must be declared in a
n enclosing class
```

25.7.5 static 필드 정의 제안

- TC39 프로세스의 stage 3에서 “Static class features”가 제안되었다.
- static public 필드, static private 필드, static private 메서드를 정의할 수 있다.

```
class MyMath {
  // static public 필드 정의
  static PI = 22 / 7;
```

```
  // static private 필드 정의
  static #num = 10;

  // static 메서드
  static increment() {
    return ++MyMath.#num;
  }
}

console.log(MyMath.PI); // 3.142857142857143
console.log(MyMath.increment()); // 11
```

25.8 상속에 의한 클래스 확장

25.8.1 클래스 상속과 생성자 함수 상속

- 상속에 의한 클래스 확장은 기존 클래스를 상속받아 새로운 클래스를 확장하여 정의하는 것

```
class Animal {
  constructor(age, weight) {
    this.age = age;
    this.weight = weight;
  }

  eat() { return 'eat'; }

  move() { return 'move'; }
}

// 상속을 통해 Animal 클래스를 확장한 Bird 클래스
class Bird extends Animal {
  fly() { return 'fly'; }
}

const bird = new Bird(1, 5);
```

```
console.log(bird); // Bird {age: 1, weight: 5}
console.log(bird instanceof Bird); // true
console.log(bird instanceof Animal); // true

console.log(bird.eat());  // eat
console.log(bird.move()); // move
console.log(bird.fly());  // fly
```

25.8.2 extends 키워드

- 서브 클래스 (subclass)
  - 상속을 통해 확장된 클래스
  - 파생 클래스 또는 자식 클래스라고도 한다.
- 수퍼 클래스 (super-class)
  - 서브 클래스에게 상속된 클래스
  - 베이스 클래스 또는 부모 클래스라고도 한다.
- 수퍼클래스와 서브클래스는 extends 키워드를 통해 상속 관계를 결정한다.
- 동시에 클래스 간의 프로토타입 체인도 생성하여

  프로토타입 메서드, 정적 메서드 모두 상속이 가능

25.8.3 동적 상속

- extends 키워드 앞에는 반드시 클래스가 와야한다

```
// 생성자 함수
function Base(a) {
  this.a = a;
}

// 생성자 함수를 상속받는 서브클래스
class Derived extends Base {}

const derived = new Derived(1);
console.log(derived); // Derived {a: 1}
```

- extends 키워드 다음에는 클래스 뿐만 아니라 [[Constructor]] 내부 메서드를 갖는 함수 객체로

  평가될 수 있는 모든 표현식이 올 수 있다.

```
function Base1() {}

class Base2 {}

let condition = true;

// 조건에 따라 동적으로 상속 대상을 결정하는 서브클래스
class Derived extends (condition ? Base1 : Base2) {}

const derived = new Derived();
console.log(derived); // Derived {}

console.log(derived instanceof Base1); // true
console.log(derived instanceof Base2); // false
```

25.8.4 서브클래스의 constructor

- 서브클래스의 constructor 역시 생략하면 빈 객체가 생성된다.
- 프로퍼티가 있는 인스턴스를 생성하려면 서브클래스 내부의 constructor에서

  프로퍼티를 추가해야 한다.

25.8.5 super 키워드

- super 호출

  super 호출 시 수퍼클래스의 constructor (super-constructor)를 호출한다.

```
// 수퍼클래스
class Base {
  constructor(a, b) {
    this.a = a;
    this.b = b;
  }
}
```

```
// 서브클래스
class Derived extends Base {
  // 다음과 같이 암묵적으로 constructor가 정의된다.
  // constructor(...args) { super(...args); }
}

const derived = new Derived(1, 2);
console.log(derived); // Derived {a: 1, b: 2}
```

- 서브 클래스에서 constructor를 사용하려면 반드시 super를 호출해야 한다.

```
class Base {}

class Derived extends Base {
  constructor() {
    // ReferenceError: Must call super constructor in deriv
ed class before accessing 'this' or returning from derived
constructor
    console.log('constructor call');
  }
}

const derived = new Derived();
```

- 서브 클래스의 constructor에서 super를 호출하기 이전에는 this를 참조할 수 없다.

```
class Base {}

class Derived extends Base {
  constructor() {
    // ReferenceError: Must call super constructor in deriv
ed class before accessing 'this' or returning from derived
constructor
    this.a = 1;
    super();
  }
}
```

```
const derived = new Derived(1);
```

- super는 반드시 서브 클래스의 constructor에서만 호출한다.

```
class Base {
  constructor() {
    super(); // SyntaxError: 'super' keyword unexpected her
e
  }
}

function Foo() {
  super(); // SyntaxError: 'super' keyword unexpected here
}
```

- super 참조

  메서드 내에서 super를 참조하면 수퍼클래스의 메서드를 호출할 수 있다.

- super 참조를 통해 수퍼클래스의 메서드를 참조하려면 super가 수퍼 클래스의 prototype 프로퍼티에 바인딩된 프로토타입을 참조할 수 있어야 한다.

```
// 수퍼클래스
class Base {
  constructor(name) {
    this.name = name;
  }

  sayHi() {
    return `Hi! ${this.name}`;
  }
}

class Derived extends Base {
  sayHi() {
    // __super는 Base.prototype을 가리킨다.
    const __super = Object.getPrototypeOf(Derived.prototyp
e);
```

```
    return `${__super.sayHi.call(this)} how are you doing?
`;
  }
}
```

```
/*
[[HomeObject]]는 메서드 자신을 바인딩하고 있는 객체를 가리킨다.
[[HomeObject]]를 통해 메서드 자신을 바인딩하고 있는 객체의 프로토타입
을 찾을 수 있다.
예를 들어, Derived 클래스의 sayHi 메서드는 Derived.prototype에 바
인딩되어 있다.
따라서 Derived 클래스의 sayHi 메서드의 [[HomeObject]]는 Derived.
prototype이고
이를 통해 Derived 클래스의 sayHi 메서드 내부의 super 참조가 Base.p
rototype으로 결정된다.
따라서 super.sayHi는 Base.prototype.sayHi를 가리키게 된다.
*/
super = Object.getPrototypeOf([[HomeObject]])
```

- ES6의 메서드 축약 표현으로 정의된 함수만이 super 참조가 가능하다.

```
const obj = {
  // foo는 ES6의 메서드 축약 표현으로 정의한 메서드다. 따라서 [[Home
Object]]를 갖는다.
  foo() {},
  // bar는 ES6의 메서드 축약 표현으로 정의한 메서드가 아니라 일반 함수
다.
  // 따라서 [[HomeObject]]를 갖지 않는다.
  bar: function () {}
};
```

```
const base = {
  name: 'Lee',
  sayHi() {
    return `Hi! ${this.name}`;
  }
};
```

```
const derived = {
  __proto__: base,
  // ES6 메서드 축약 표현으로 정의한 메서드다. 따라서 [[HomeObjec
t]]를 갖는다.
  sayHi() {
    return `${super.sayHi()}. how are you doing?`;
  }
};

console.log(derived.sayHi()); // Hi! Lee. how are you doin
g?
```

```
// 수퍼클래스
class Base {
  static sayHi() {
    return 'Hi!';
  }
}

// 서브클래스
class Derived extends Base {
  static sayHi() {
    // super.sayHi는 수퍼클래스의 정적 메서드를 가리킨다.
    return `${super.sayHi()} how are you doing?`;
  }
}

console.log(Derived.sayHi()); // Hi! how are you doing?
```

25.8.6 상속 클래스의 인스턴스 생성 과정

1. 서브 클래스의 super 호출

   자바스크립트 엔진은 수퍼 클래스와 서브 클래스 구분을 위해 내부 슬롯 `[[ConstructorKind]]`를 갖는다.

   아무것도 상속받지 않는 클래스는 “base”로, 상속을 받는 클래스는 “derived”로 설정된다.

2. 수퍼클래스의 인스턴스 생성과 this 바인딩

   super를 통해 수퍼 클래스의 인스턴스가 생성되지만, new 연산자와 함께 호출된 클래스는 서브 클래스이다. 따라서 인스턴스는 서브 클래스가 생성한 것으로 처리된다.

```
// 수퍼클래스
class Rectangle {
  constructor(width, height) {
    // 암묵적으로 빈 객체, 즉 인스턴스가 생성되고 this에 바인딩된
다.
    console.log(this); // ColorRectangle {}
    // new 연산자와 함께 호출된 함수, 즉 new.target은 ColorRec
tangle이다.
    console.log(new.target); // ColorRectangle
...
```

```
// 수퍼클래스
class Rectangle {
  constructor(width, height) {
    // 암묵적으로 빈 객체, 즉 인스턴스가 생성되고 this에 바인딩된
다.
    console.log(this); // ColorRectangle {}
    // new 연산자와 함께 호출된 함수, 즉 new.target은 ColorRec
tangle이다.
    console.log(new.target); // ColorRectangle

    // 생성된 인스턴스의 프로토타입으로 ColorRectangle.prototyp
e이 설정된다.
    console.log(Object.getPrototypeOf(this) === ColorRec
tangle.prototype); // true
    console.log(this instanceof ColorRectangle); // true
    console.log(this instanceof Rectangle); // true
...
```

3. 수퍼클래스의 인스턴스 초기화

   수퍼클래스의 constructor가 실행되어 this에 바인딩되어있는 인스턴스 프로퍼티를 초기화한다.

```
// 수퍼클래스
class Rectangle {
  constructor(width, height) {
    // 암묵적으로 빈 객체, 즉 인스턴스가 생성되고 this에 바인딩된
다.
    console.log(this); // ColorRectangle {}
    // new 연산자와 함께 호출된 함수, 즉 new.target은 ColorRec
tangle이다.
    console.log(new.target); // ColorRectangle

    // 생성된 인스턴스의 프로토타입으로 ColorRectangle.prototyp
e이 설정된다.
    console.log(Object.getPrototypeOf(this) === ColorRec
tangle.prototype); // true
    console.log(this instanceof ColorRectangle); // true
    console.log(this instanceof Rectangle); // true

    // 인스턴스 초기화
    this.width = width;
    this.height = height;

    console.log(this); // ColorRectangle {width: 2, heig
ht: 4}
  }
...
```

4. 서브클래스 constructor로의 복귀와 this 바인딩

   super가 반환한 인스턴스가 this에 바인딩되고 서브클래스는 이를 그대로 사용한다.

```
// 서브클래스
class ColorRectangle extends Rectangle {
  constructor(width, height, color) {
    super(width, height);

    // super가 반환한 인스턴스가 this에 바인딩된다.
    console.log(this); // ColorRectangle {width: 2, heig
```

```
ht: 4}
...
```

즉, super가 호출되지 않으면 this 바인딩이 불가능하다.

5. 서브클래스의 인스턴스 초기화

   super 호출 이후 서브클래스의 constructor 쪽의 인스턴스가 초기화된다.

6. 인스턴스 반환

   모든 처리가 끝난 후 인스턴스가 바인딩된 this가 암묵적으로 반환된다.

```
// 서브클래스
class ColorRectangle extends Rectangle {
  constructor(width, height, color) {
    super(width, height);

    // super가 반환한 인스턴스가 this에 바인딩된다.
    console.log(this); // ColorRectangle {width: 2, heig
ht: 4}

    // 인스턴스 초기화
    this.color = color;

    // 완성된 인스턴스가 바인딩된 this가 암묵적으로 반환된다.
    console.log(this); // ColorRectangle {width: 2, heig
ht: 4, color: "red"}
  }
...
```

25.8.7 표준 빌트인 생성자 함수 확장

- String, Number, Array와 같은 표준 빌트인 객체 역시

  [[Construct]] 내부 메서드를 갖는 생성자 함수

- 따라서 extends 키워드를 통한 확장이 가능하다.

```
// Array 생성자 함수를 상속받아 확장한 MyArray
class MyArray extends Array {
  // 중복된 배열 요소를 제거하고 반환한다: [1, 1, 2, 3] => [1, 2,
```

```
3]
  uniq() {
    return this.filter((v, i, self) => self.indexOf(v) ===
i);
  }

  // 모든 배열 요소의 평균을 구한다: [1, 2, 3] => 2
  average() {
    return this.reduce((pre, cur) => pre + cur, 0) / this.l
ength;
  }
}

const myArray = new MyArray(1, 1, 2, 3);
console.log(myArray); // MyArray(4) [1, 1, 2, 3]

// MyArray.prototype.uniq 호출
console.log(myArray.uniq()); // MyArray(3) [1, 2, 3]
// MyArray.prototype.average 호출
console.log(myArray.average()); // 1.75
```